

- 5 typedef unsigned int GLenum
- 5 typedef unsigned int GLbitfield
- 5 typedef int GLint
- 5 typedef int GLsizei
- 5 typedef float GLfloat
- 5 typedef float GLclampf
- 5 typedef double GLdouble
- 5 void glutInit(int *argc, char **argv)
 - 5 GLUTライブラリを初期化します.
 - 5 「argc」と「argv」はmain関数の引数, すなわちコマンドライン引数を渡します. これらの引数は, コマンドラインのオプション指定時に用いられます.
- 5 void glutInitDisplayMode(unsigned int mode)
 - 5 ディスプレイの表示モードを設定します.
 - 5 「glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH)」のように書くと, 「RGBAカラーモデル」で「ダブルバッファ」を使い, 「デプスバッファ」も使うという指定になります.
- 5 void glutInitWindowSize(int width, int height)
 - 5 ウィンドウの初期サイズを設定します.
 - 5 「width」はウィンドウの幅, 「height」はウィンドウの高さになります.
- 5 void glutInitWindowPosition(int x, int y)
 - 5 ウィンドウの左上の位置を指定する. 引数は共にピクセル値.
- 5 int glutCreateWindow(char *title)
 - 5 ウィンドウを生成する. 引数はそのウィンドウの名前となる.
- 5 void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)
 - 5 「glClearColor(GL_COLOR_BUFFER_BIT)」でウィンドウを塗りつぶす際の色を指定します.
 - 5 「red」「green」「blue」はそれぞれ「赤」「緑」「青色」の成分の強さを示すGLclampf型(float型と等価)の値で, 0~1の間の値をもちます. 1が最も明るく, この3つに(0,0,0)を指定すれば「黒色」になり, (1,1,1)を指定すれば「白色」になります.
 - 5 最後の「alpha」は「α値」と呼ばれ, OpenGLでは不透明度として扱われます(0で透明, 1で不透明). ここではとりあえず「1」にしておいてください.
- 5 void glutMainLoop(void)
 - 5 GLUTのイベントが発生するまで, 待機状態になります.
- 5 void glutSwapBuffers(void)
 - 5 描画の最後で記述する. この関数が実行されると, バックバッファの内容がフロントバッファに転送される.
- 5 void glClear(GLbitfield mask)
 - 5 「mask」に指定したバッファのビットを初期化します.
 - 5 「glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)」と指定すると「カラーバッファ」と「Zバッファ」が初期化されます.
- 5 void glEnable(GLenum cap)
 - 5 GLenum型の引数「cap」に指定した機能を使用可能にします.
 - 5 「glEnable(GL_DEPTH_TEST)」を実行すると, それ以降「Zバッファ」を使います.
 - 5 「glEnable(GL_LIGHTING)」を実行すると, それ以降「陰影付け」の計算をします.
 - 5 「glEnable(GL_LIGHT0)」を実行すると, 0番目の光源を点灯します.

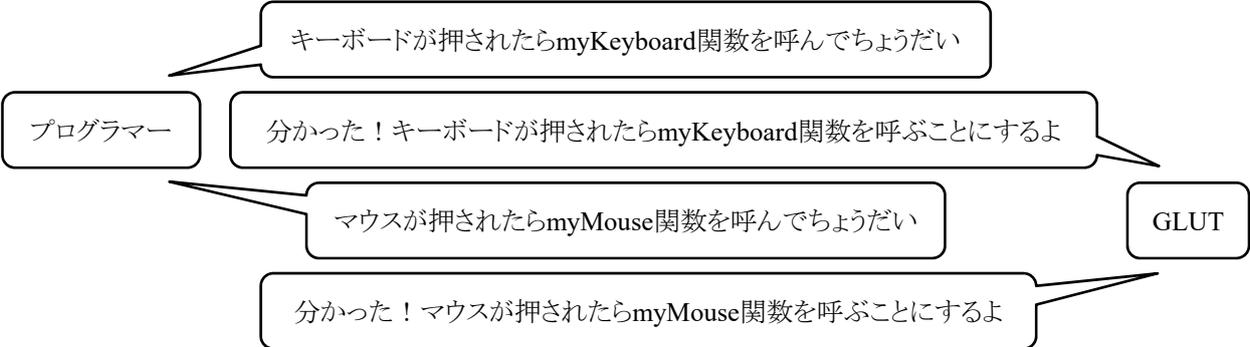
- ❏ **void glutDisplayFunc(void (*)(void))**
 - ❏ 引数は開いたウィンドウ内に描画する関数へのポインタです。ウィンドウが開かれたり、他のウィンドウによって隠されたウィンドウが再び現われたりしてウィンドウを再描画する必要があるときに、この関数が実行されます。したがって、この関数内で図形表示を行います。
- ❏ **void glutReshapeFunc(void (*)(int width, int height))**
 - ❏ 引数には、ウィンドウがリサイズされたときに実行する関数のポインタを与えます。この関数の引数にはリサイズ後のウィンドウの幅と高さが渡されます。
- ❏ **void glutTimerFunc(unsigned int millis, void (*)(int value), int value)**
 - ❏ 指定された時間に呼び出されるコールバック関数を登録します。異なる時間のコールバック関数を複数用意できます。
 - ❏ 「millis」は呼び出される時間をミリ秒で指定します。少なくとも「millis」ミリ秒後にコールされるようになります。
 - ❏ 第3引数の「value」は登録したタイマーコールバック関数に渡されます。
- ❏ **void glutKeyboardFunc(void (*)(unsigned char key, int x, int y))**
 - ❏ 引数には、キーがタイプされたときに実行する関数のポインタを与えます。この関数の引数「key」には、タイプされたキーのASCIIコードが渡されます。また、「x」と「y」にはキーがタイプされたときのマウスの位置が渡されます。
- ❏ **void glutPostRedisplay(void)**
 - ❏ ウィンドウを再描画します。glutDisplayFunc()で登録したコールバック関数が呼び出されます。
- ❏ **void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)**
 - ❏ 「ビューポート」を設定します。「ビューポート」とは、開いたウィンドウの中で、実際に描画される領域のことをいいます。正規化デバイス座標系の2点(-1,-1), (1,1)を結ぶ線分を対角線とする矩形領域がここに表示されます。最初の2つのGLint型 (int型と等価)の引数「x,y」にはその領域の左下隅の位置、後の2つのGLsizei型 (int型と等価)の「width」と「height」には、それぞれ幅と高さをデバイス座標系の値、すなわちディスプレイ上の画素数で指定します。
glutReshapeFuncで指定されたコールバック関数の引数「width,height」にはそれぞれウィンドウの幅と高さが入っていますから、glViewport(0,0,width,height)はリサイズ後のウィンドウの全面を表示領域に使うことになります。

- 5 void glMatrixMode(GLenum mode)
 - 5 設定する変換行列を指定します。引数「mode」が「GL_MODELVIEW」なら「モデルビュー変換行列」を指定し、「GL_PROJECTION」なら「透視変換行列」を指定します。
- 5 void glLoadIdentity(void)
 - 5 これは変換行列を初期化します。座標変換の合成は行列の積で表されますから、この関数を使って変換行列に初期値として単位行列を設定します。
- 5 void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)
 - 5 変換行列に透視変換の行列を乗じます。
 - 5 最初の引数「fovy」はカメラの画角であり、「度」で表します。これが大きいほど広角(透視が強くなり、絵が小さくなります)になり、小さいほど望遠レンズになります。
 - 5 2つ目の引数「aspect」は画面のアスペクト比(縦横比)であり、「1」であればビューポートに表示される図形のx方向とy方向のスケールが等しくなります。
 - 5 3つ目の引数「zNear」と4つ目の引数「zFar」は表示する奥行き方向の範囲で、「zNear」は手前(前方面)、zFarは後方(後方面)の位置を示します。この間にある図形が描画されます。
- 5 void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz)
 - 5 この最初の3つの引数「eyex, eyey, eyez」は視点の位置、次の3つの引数「centerx, centery, centerz」は目標の位置、最後の3つの引数「upx, upy, upz」は、ウィンドウに表示される画像の「上」の方向を示すベクトルです。
- 5 void glPushMatrix(void)
 - 5 glMatrixMode()で指定している現在の変換行列を保存します。
- 5 void glPopMatrix(void)
 - 5 glPopMatrix()で保存した変換行列を復帰します。したがって、「glPushMatrix()」を呼び出した後、glTranslated()やglRotated()あるいはgluLookAt()などを使って変換行列を変更しても、「glPopMatrix()」を呼び出すことによって、それ以前の変換行列に戻すことができます。
- 5 void glTranslated(GLdouble x, GLdouble y, GLdouble z)
 - 5 変換行列に平行移動の行列を乗じます。引数はいずれもGLdouble型で、3つの引数「x」「y」「z」には現在の位置からの相対的な移動量を指定します。
- 5 void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z)
 - 5 変換行列に回転の行列を乗じます。引数はいずれもGLdouble型で、1つ目の引数「angle」は回転角、残りの3つの引数「x」「y」「z」は回転軸の方向ベクトルです。
 - 5 回転角「angle」の単位は度(°)(degree)です。C言語のcos関数などの引数の単位はラジアンなので注意してください。
- 5 void glBegin(GLenum mode)
 - 5 void glEnd(void)
 - 5 「glBegin」と「glEnd」の間に指定した頂点座標を使って、描画を行います。
 - 5 描画内容は「mode」に指定します。「mode」には「GL_QUADS」などが指定できます。
- 5 void glVertex3d(GLdouble x, GLdouble y, GLdouble z)
 - 5 3次元の座標値を設定します。
 - 5 引数はGLdouble型の(x, y, z)で指定します。
- 5 void glNormal3d(GLdouble nx, GLdouble ny, GLdouble nz)
 - 5 単位法線ベクトルを設定します。
 - 5 引数はGLdouble型の(nx, ny, nz)で指定します。
- 5 void glMaterialfv(GLenum face, GLenum pname, GLfloat *params)
 - 5 表面属性を定義する。
 - 5 「face」に「GL_FRONT」を指定すると、ポリゴンの表面のみに属性を設定します。
 - 5 「pname」に「GL_DIFFUSE」を指定すると、「params」でfloat型の配列を指定することで、材質の拡散RGBA値を設定できます。その際はglMaterialfvを使います。

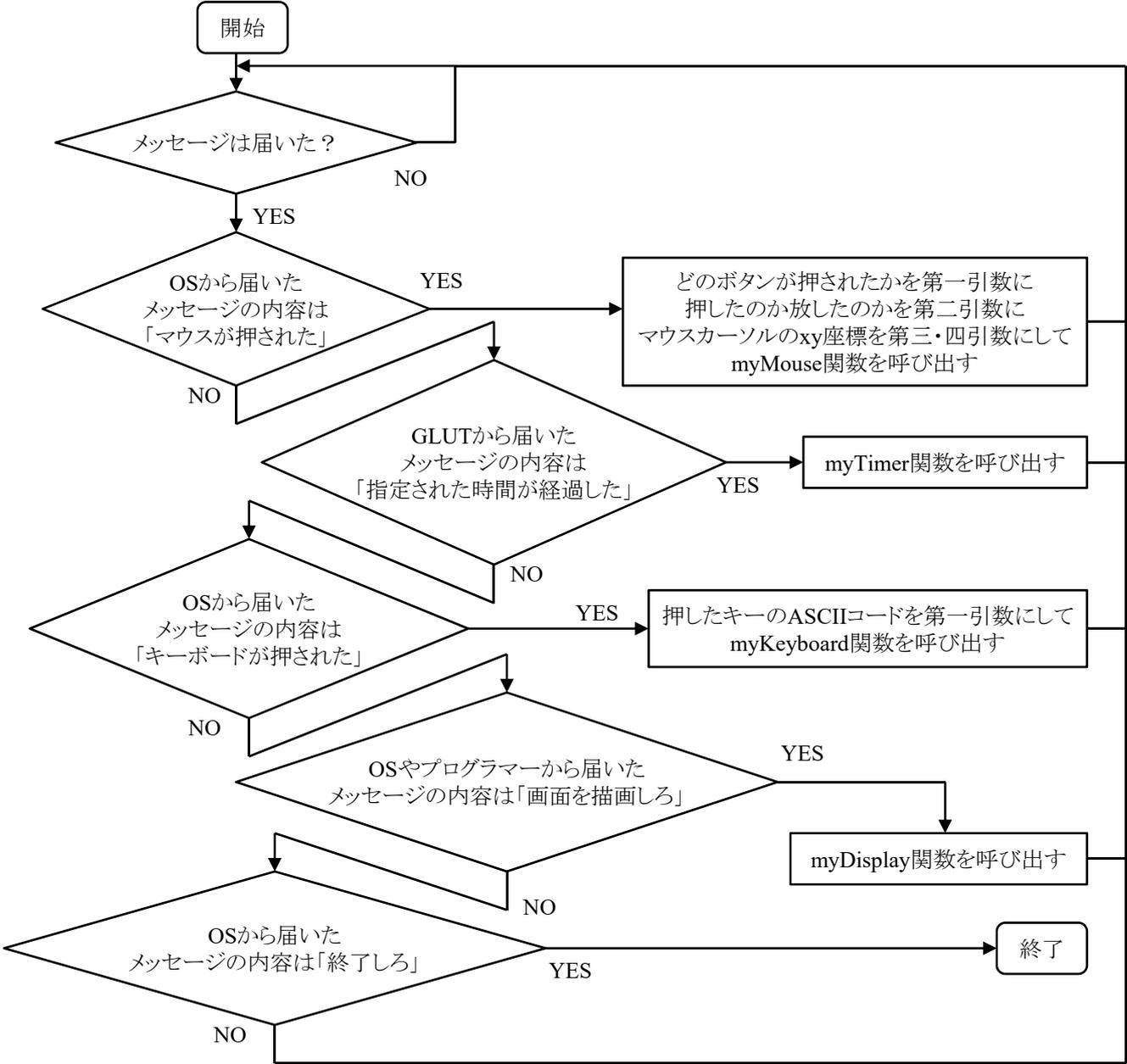
イベントドリブン型プログラミング

- glutMainLoopは無限ループしているだけで、メッセージが届くのをひたすら待ち続けます
- メッセージを受け取ったら、メッセージに応じた処理をして、また再び無限に待機します

コールバック関数の指定



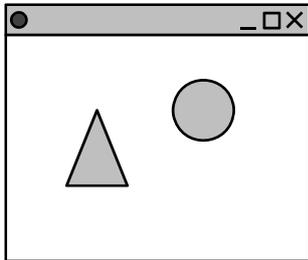
ループ中の動作の例



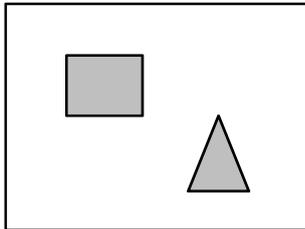
画面の描画

🔗 `glClear`で画面を消去して`glutSwapBuffers`で描画します

描画の流れ

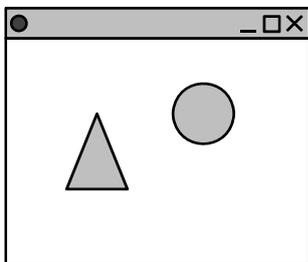


ウィンドウ表示用のバッファ



メモリ内にあるバッファ

```
glClear(GL_COLOR_BUFFER_BIT);  
画面消去
```

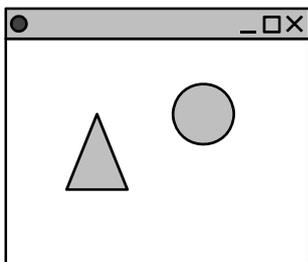


ウィンドウ表示用のバッファ

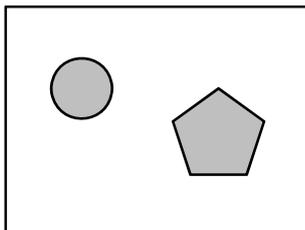


メモリ内にあるバッファ

```
glBegin~glEnd  
図形の描画
```

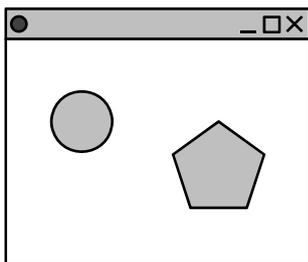


ウィンドウ表示用のバッファ

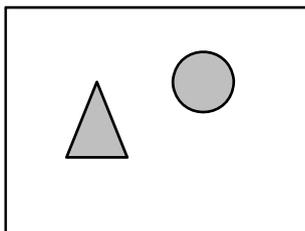


メモリ内にあるバッファ

```
glutSwapBuffers();  
ウィンドウに表示
```



ウィンドウ表示用のバッファ



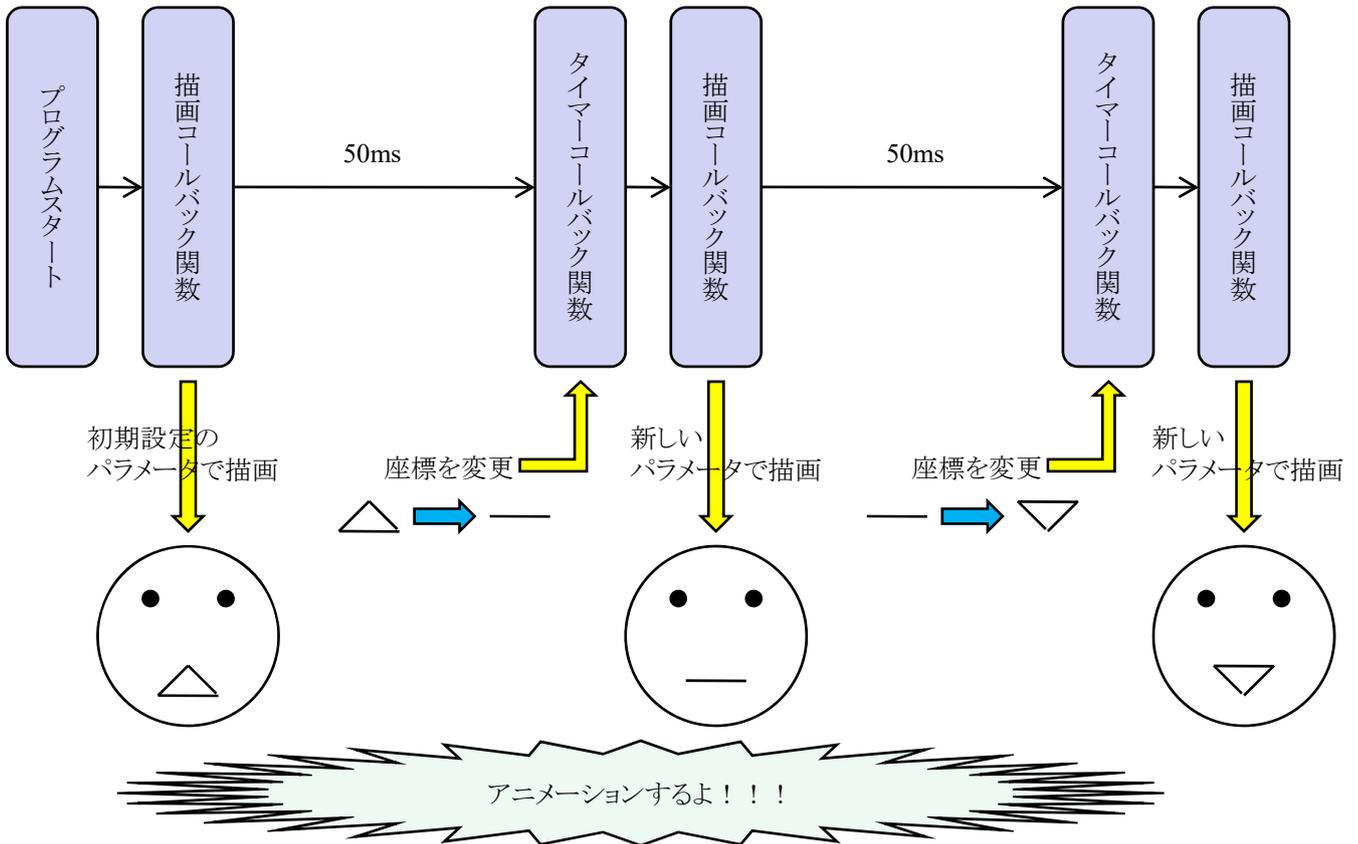
メモリ内にあるバッファ

```
void ディスプレイコールバック関数()  
{  
    // 画面消去  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    // 図形の描画  
    glBegin(GL_QUADS);  
    glVertex3d(-1.0, -1.0, 0.0);  
    glVertex3d(1.0, -1.0, 0.0);  
    glVertex3d(1.0, 1.0, 0.0);  
    glVertex3d(-1.0, 1.0, 0.0);  
    glEnd();  
  
    // ウィンドウに表示  
    glutSwapBuffers();  
}
```

タイマー

- void glutTimerFunc(unsigned int msec, void (*func)(int value), int value)
 - 指定された時間に呼び出されるコールバック関数を登録します。異なる時間のコールバック関数を複数用意できます。
 - 「msec」は呼び出される時間をミリ秒で指定します。少なくとも「msec」ミリ秒後にコールされるようになります。
 - 第3引数の「value」は登録したタイマーコールバック関数に渡されます。
- void glutPostRedisplay(void)
 - ウィンドウ内の再描画を行います。より正確には、現在のウィンドウをマークして、「display関数」を呼び出します。

```
glutTimerFunc(50,myTimer,1); // 50ミリ秒後にmyTimer関数を呼べ  
void myTimer(int value) {  
    // 描画オブジェクトの座標や各種パラメータを変更する処理を行う  
    glutTimerFunc(50,myTimer,1); // 再び50ミリ秒後にmyTimer関数を呼べ  
    glutPostRedisplay(); // ウィンドウの内容を再描画しろ(新しい座標やパラメータで描画)  
}
```



点, 線, ポリゴンの描画

点, 線, ポリゴンを描くのに, 次のようにglBegin()とglEnd()および, glVertex*()を用いる.

```
glBegin(mode);
  glVertex*(p0);
  glVertex*(p1);
  .....
  glVertex*(pn);
glEnd();
```

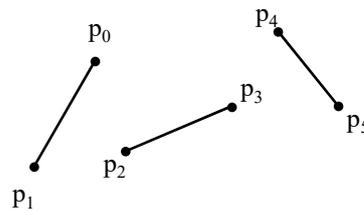
modeは描く図形の種類を指定し, p0, p1, ..., pnは座標位置を意味する.

modeの種類

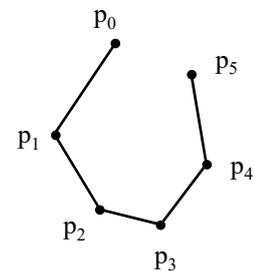
- GL_LINE_STRIP 最初の頂点から最後の頂点まで線分を連結して描画する.
- GL_LINES 二つの頂点を結んだ直線を生成する.
- GL_TRIANGLES 三つ一組の頂点を, それぞれ独立した三角形として描画する.
- GL_QUADS 四つ一組の頂点を, それぞれ独立した四角形として描画する.
- GL_POLYGON 単独の凸ポリゴンを描画する.

ポリゴン描写の注意点

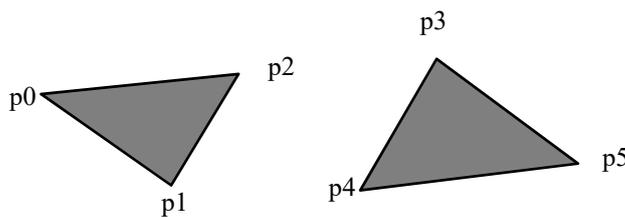
- ポリゴン (polygon) とは多角形の意味
- 頂点座標は反時計回りに設定すること
 - ポリゴンの表面と裏面を区別するため
 - 視点から見て頂点座標が反時計回りに配置されているポリゴンは表面, 時計回りの場合は裏面と約束されている



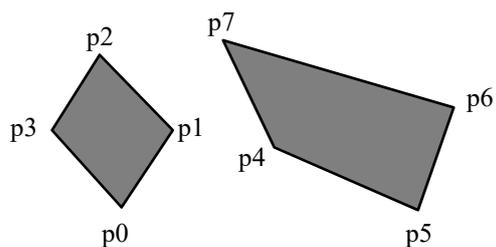
GL_LINES



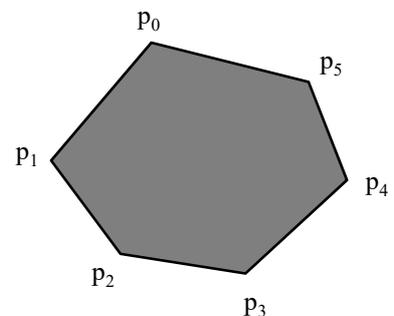
GL_LINE_STRIP



GL_TRIANGLES



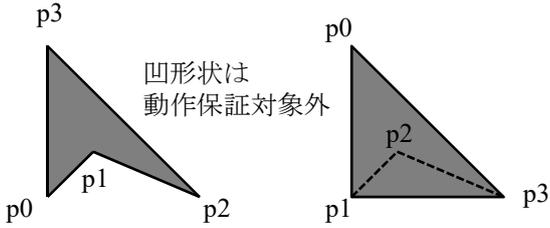
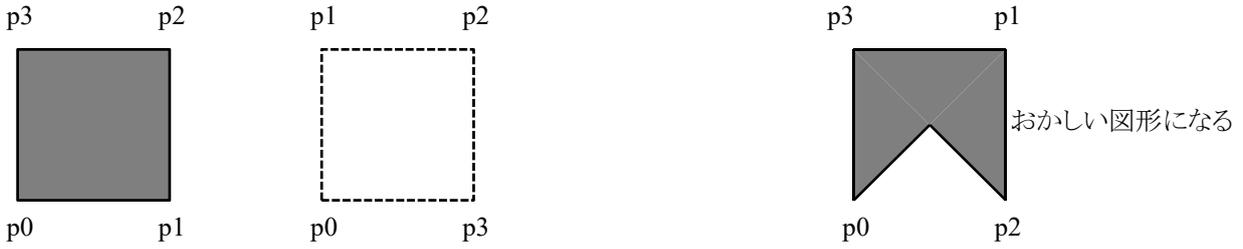
GL_QUADS



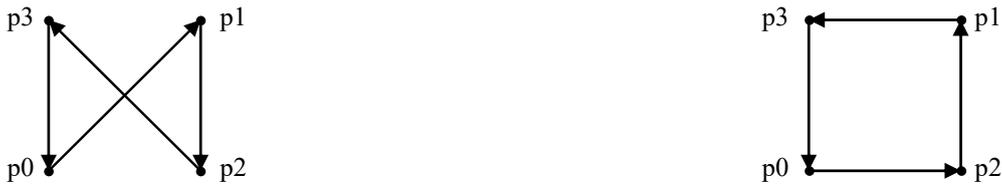
GL_POLYGON

四角形の頂点の順番

4つの頂点の順番によって表示される図形が異なる



右の例は、 $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow p_3$ の順番だからおかしくなるのであって、 $p_0 \rightarrow p_2 \rightarrow p_1 \rightarrow p_3$ の順番で `glVertex?d`関数を呼び出せば、正しく描画される

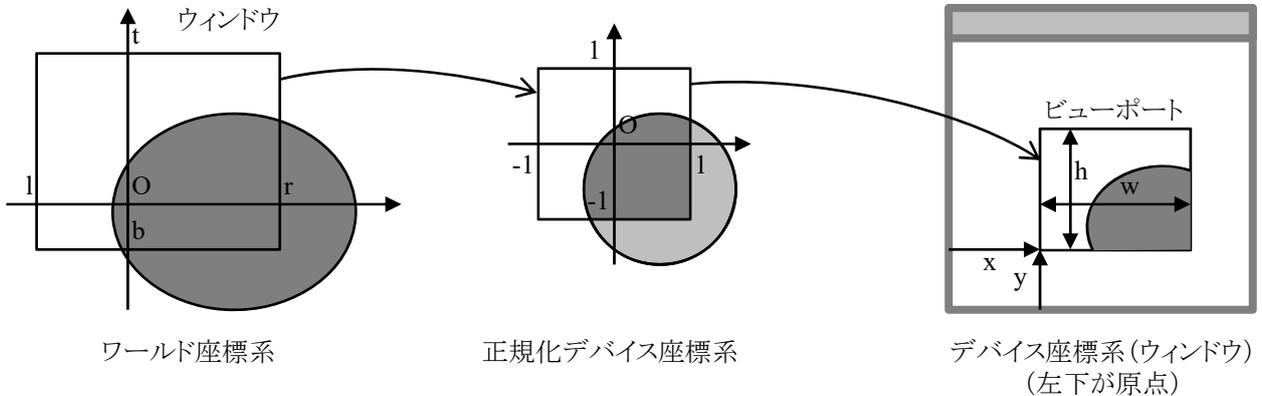


OpenGLの仕様とは異なる使い方をした場合(この場合、凸多角形にしか対応していないという仕様を無視して、非凸多角形を描画しようとした場合)の動作は保証されない。

<http://monobook.org/wiki/OpenGL>

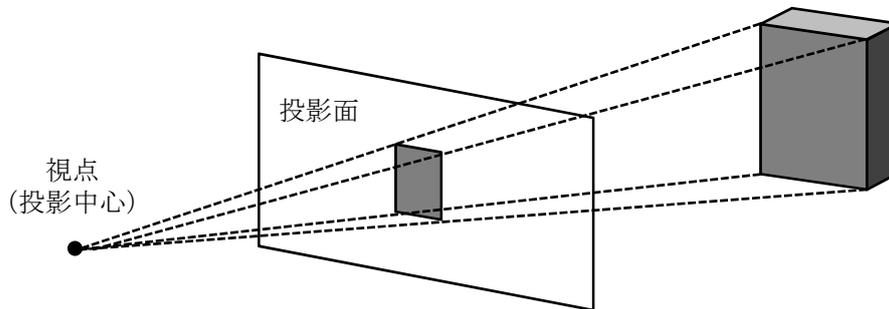
通常、凹多角形は三角形に分割して、凸多角形で表現するのが普通(検索キーワード: 三角形分割)

ウィンドウとビューポート



透視投影 (perspective projection)

- 視点と投影面を指定
- 人間の見え方に近い



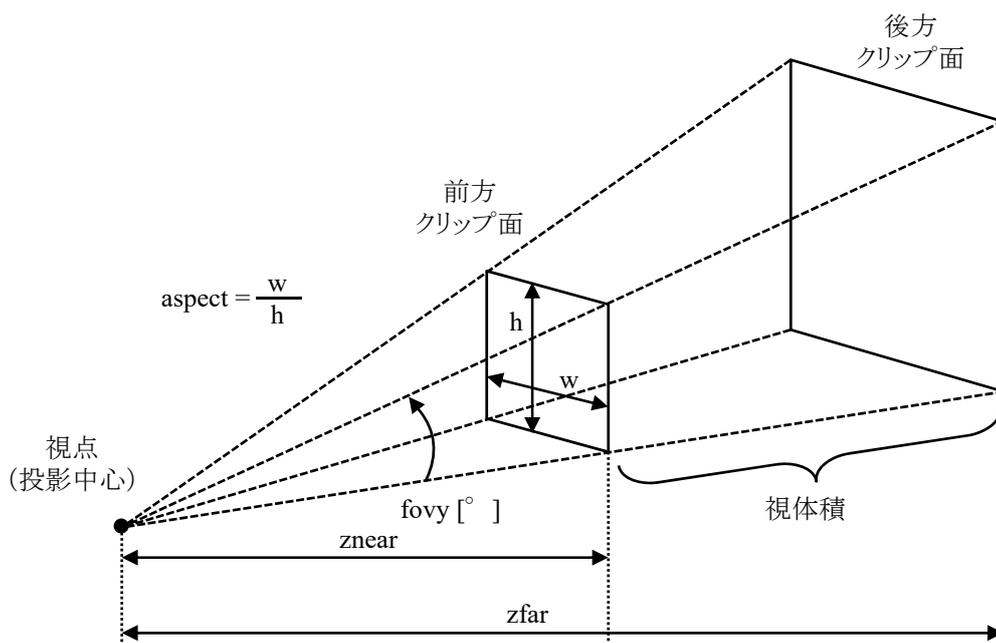
視体積

- オブジェクトが見える領域を視体積(view volume)という
- この錐台の外にあるオブジェクトは表示されない

関数

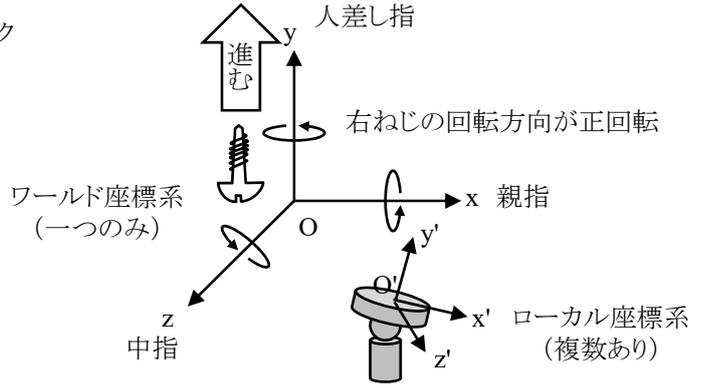
- `void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)`
 - 変換行列に透視変換の行列を乗じます。
 - 最初の引数「fovy」はカメラの画角であり、「度」で表します。これが大きいほど広角(透視が強くなり、絵が小さくなります)になり、小さいほど望遠レンズになります。
 - 2つ目の引数「aspect」は画面のアスペクト比(縦横比)であり、「1」であればビューポートに表示される図形のx方向とy方向のスケールが等しくなります。
 - 3つ目の引数「zNear」と4つ目の引数「zFar」は表示する奥行き方向の範囲で、「zNear」は手前(前方面)、zFarは後方(後方面)の位置を示します。この間にある図形が描画されます。

透視投影の視体積



右手系

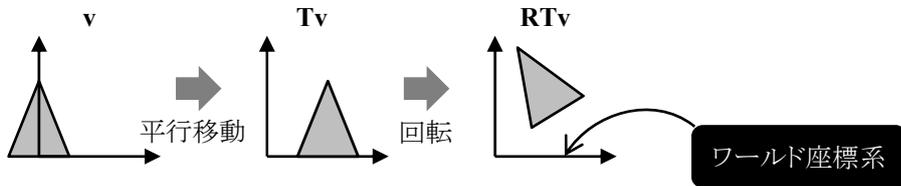
- 📌 ワールド座標系: ユーザが描く図形やオブジェクト全てに対して共通な座標系である
- 📌 ローカル座標系: 各オブジェクト固有の座標系であり, ワールド座標系の中に複数存在する



モデリング変換

- 📌 OpenGLは, オブジェクトの平行移動(translation), 回転(rotation), スケーリング(scaling)の三つのモデリング変換を用意している
- 📌 OpenGLは, 2次元や3次元の頂点を扱うが, 内部では統一的に4次元ベクトル(x, y, z, w)を用いる. これを同次座標という.
- 📌 OpenGLでは, 点 $v=(x\ y\ z\ 1)^T$ を点 $v'=(x'\ y'\ z'\ 1)^T$ に変換する一般式は $v'=Tv$ で表される
- 📌 T は 4×4 の行列で, 変換行列を表す
- 📌 変換行列には, モデルビュー行列(model view matrix)と投影行列(projection matrix)がある
- 📌 OpenGLでは, ローカル座標系の考え方をすると, プログラムの処理順序に合う

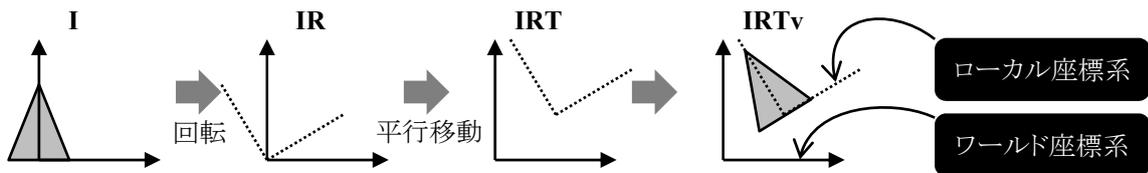
ワールド座標系の考え方



```

glLoadIdentity();
glRotate*();
glTranslate*();
glWireCone();
    
```

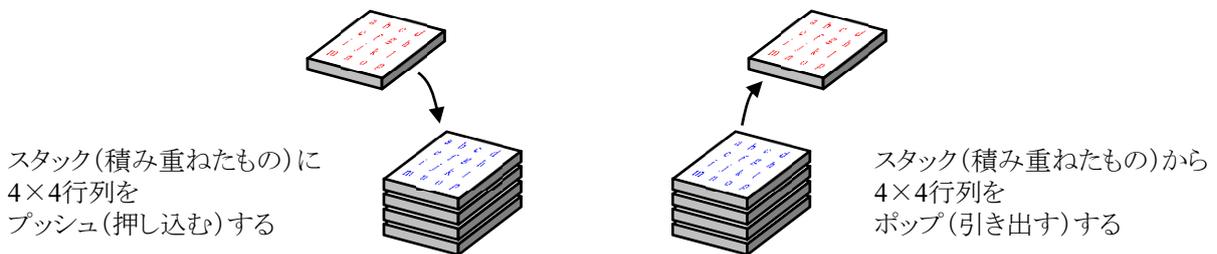
ローカル座標系の考え方



スタック

- 📌 OpenGLでは, 変換行列をスタック領域に格納することができる

PUSH (プッシュ) **POP (ポップ)**



平行移動

glTranslated関数(座標系を変更する場合)

- 📌 glTranslated関数でローカル座標系を平行移動できるので、ローカル座標系を移動させたあと、図形の描画命令を書けばいいわけです
 - 📌 ローカル座標系から見れば図形は移動していませんが、ワールド座標系から見れば図形が移動しているかのように見えます

例1

```
glPushMatrix();  
glTranslated(posx, posy, 0.5);  
glutSolidSphere(1.0, 10, 10);  
glPopMatrix();
```

例2

```
glPushMatrix();  
glTranslated(posx, posy, 1.9);  
myHuman(timestep, cycle);  
glPopMatrix();
```

例3

```
glPushMatrix();  
glTranslated(posx, posy, 0.1);  
glBegin(GL_QUADS);  
glVertex3d(-0.5, -0.5, 0.0);  
glVertex3d(+0.5, -0.5, 0.0);  
glVertex3d(+0.5, +0.5, 0.0);  
glVertex3d(-0.5, +0.5, 0.0);  
glEnd();  
glPopMatrix();
```

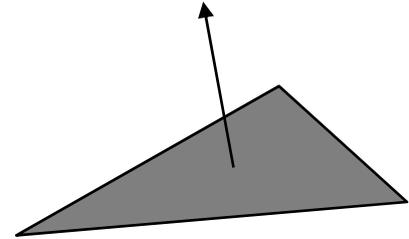
法線

- 法線は、面に垂直な方向を指しているベクトルです。OpenGLでは、各ポリゴン、または各頂点に対して法線を指定できます。
- 現在の法線は`glNormal*()`で設定します。それ以降に呼び出された`glVertex*()`で指定した頂点に現在の法線が割り当てられます。

用例

```
glBegin(GL_POLYGON);  
  glNormal3d(n0x, n0y, n0z);  
  glVertex3d(v0x, v0y, v0z);  
  glNormal3d(n1x, n1y, n1z);  
  glVertex3d(v1x, v1y, v1z);  
  glNormal3d(n2x, n2y, n2z);  
  glVertex3d(v2x, v2y, v2z);  
  glNormal3d(n3x, n3y, n3z);  
  glVertex3d(v3x, v3y, v3z);  
glEnd();
```

面の向きを表す
長さは1

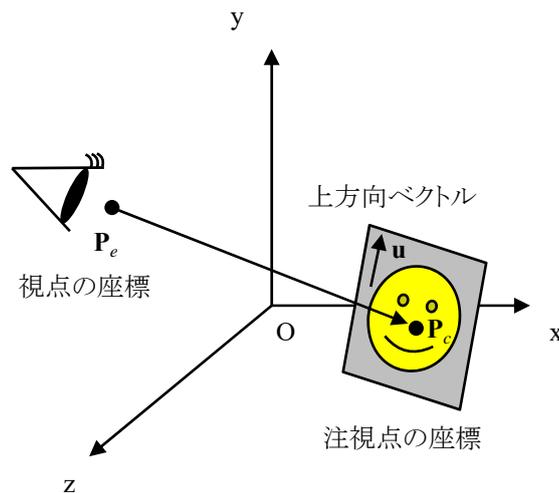


関数の説明

- `void glNormal3d(GLdouble nx, GLdouble ny, GLdouble nz);`
 - 法線ベクトルを設定します。

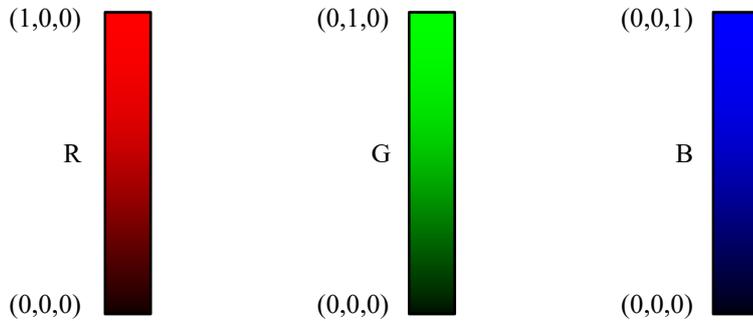
視点変換

- `void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez, GLdouble cx, GLdouble cy, GLdouble cz, GLdouble ux, GLdouble uy, GLdouble uz)`
 - この最初の3つの引数「`ex, ey, ez`」は視点の位置、次の3つの引数「`cx, cy, cz`」は目標の位置、最後の3つの引数「`ux, uy, uz`」は、ウィンドウに表示される画像の「上」の方向を示すベクトルです。



色の表現

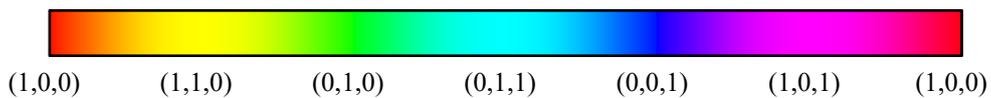
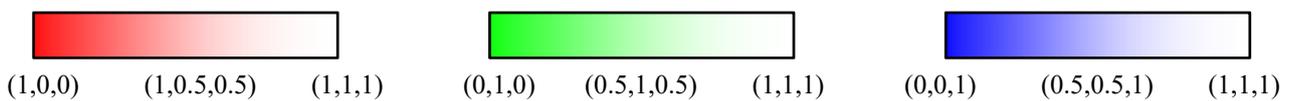
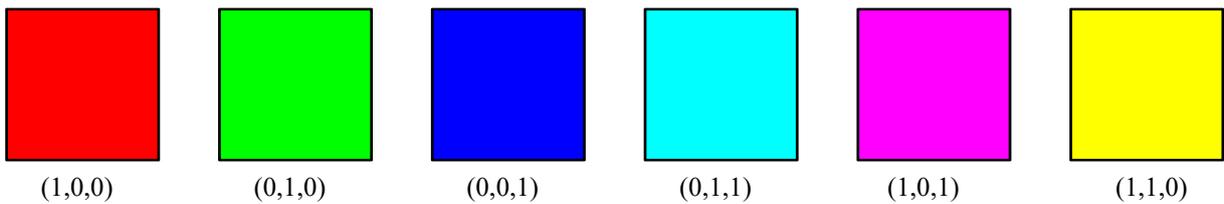
- 光の三原色で表す
- R: Red 赤, G: Green 緑, B: Blue 青
- glClearColor関数で背景色を指定, glColor3d関数で描画する図形の色を指定します
- OpenGLのこれらの関数の引数は, 浮動小数点の0~1の値で表します
 - ちなみに, 画面や画像ファイルなどの画素は通常, RGBそれぞれ8bitずつ, 合計24bitで表します. 整数で0~255の値で表します. OpenGLは0~1ですのでご注意ください.
- 0が暗くて, 1が明るいです



(R,G,B)=(0,0,0)

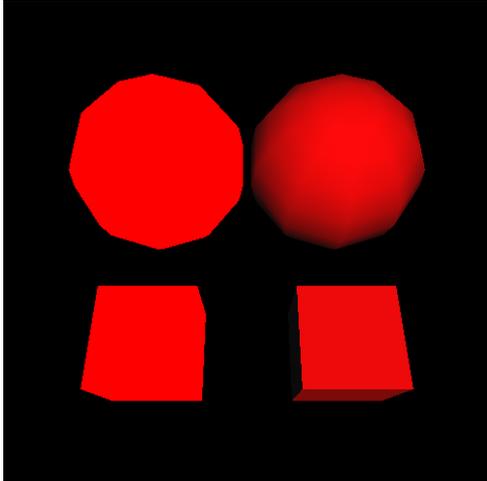
(R,G,B)=(0.5,0.5,0.5)

(R,G,B)=(1,1,1)



図形の色

←ライティングがオフの場合



←ライティングがオンの場合

glColor3d関数(ライティングしない場合)

- 📌 初期状態ではglDisable(GL_LIGHTING), すなわち、ライティング計算をしない状態になっています。
- 📌 陰影は計算されません。
- 📌 図形の色はglColor3d関数で指定します。

例1

```
glColor3d(1.0, 0.0, 0.0);  
glutSolidSphere(1.0, 10, 10);
```

例2

```
glColor3d(1.0, 0.0, 0.0);  
glBegin(GL_QUADS);  
glVertex3d(-1.0, -1.0, 0.0);  
glVertex3d(1.0, -1.0, 0.0);  
glVertex3d(1.0, 1.0, 0.0);  
glVertex3d(-1.0, 1.0, 0.0);  
glEnd();
```

glMaterialfv関数(ライティングする場合)

- 📌 glEnable(GL_LIGHTING)で、ライティング計算をする状態になります。
- 📌 glEnable(GL_LIGHT0)などで、光源を点灯する必要があります。
- 📌 光源と法線の関係を使って陰影が計算されます。
- 📌 図形には法線が正しく設定されていなければいけません。
- 📌 図形の色はglMaterialfv関数で指定します。

例1

```
float red[4] = { 1.0f, 0.0f, 0.0f, 1.0f };  
glEnable(GL_LIGHTING);  
glEnable(GL_LIGHT0);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, red);  
glutSolidSphere(1.0, 10, 10);
```

例2

```
float red[4] = { 1.0f, 0.0f, 0.0f, 1.0f };  
glEnable(GL_LIGHTING);  
glEnable(GL_LIGHT0);  
glMaterialfv(GL_FRONT, GL_DIFFUSE, red);  
glBegin(GL_QUADS);  
glNormal3d(0.0, 0.0, 1.0);  
glVertex3d(-1.0, -1.0, 0.0);  
glVertex3d(1.0, -1.0, 0.0);  
glVertex3d(1.0, 1.0, 0.0);  
glVertex3d(-1.0, 1.0, 0.0);  
glEnd();
```

光源の有効・無効化

- ☞ glEnable(GL_LIGHTING);
 - ☞ 光源による陰影を計算する機能をオンにする
- ☞ glEnable(GL_LIGHTn);
 - ☞ 番号n=0~7の光源を点灯する
- ☞ glLightModeli(GL_LIGHT_MODEL_COLOR_MATERIAL, GL_FALSE);
 - ☞ 光源が点灯された状態で描画したいシーンのOpenGL命令を実行する
- ☞ glDisable(GL_LIGHTn);
 - ☞ 番号n=0~7の光源を消灯する
- ☞ glDisable(GL_LIGHTING);
 - ☞ 光源による陰影を計算する機能をオフにする

物体表面の材質の設定

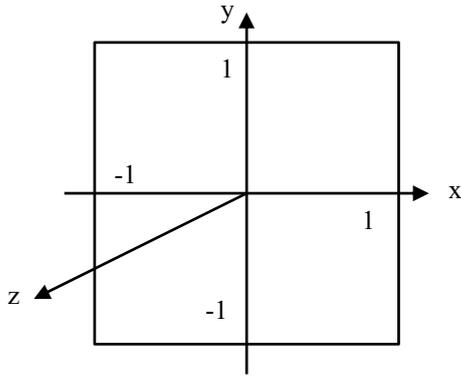
- ☞ void glMaterialfv(GLenum face, GLenum pname, GLfloat *param)
 - ☞ 表面属性を定義する.
 - ☞ faceにGL_FRONTを指定すると, ポリゴンの表面のみに属性を設定します.

pname	params	パラメータ
GL_DIFFUSE	float型の4次元配列	材質の拡散RGBA値

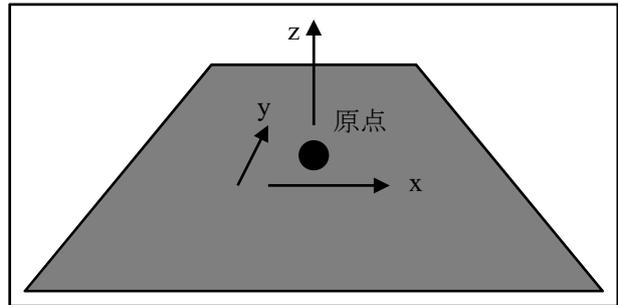
本日の課題

- 5 [課題]人間の歩行動作アニメーションを作成せよ
- 5 [やること]
 - 5 関節の角度を適切に設定する
 - 5 適切な箇所でglPushMatrixやglPopMatrixを呼び出す

ワールド座標系

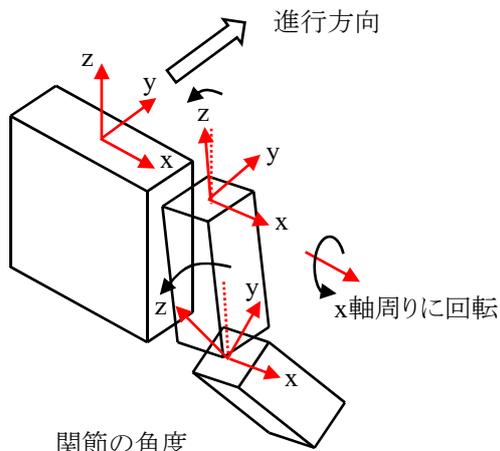


初期状態

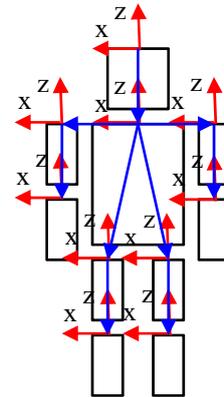


gluLookAtで設定された今回の課題の座標

ローカル座標系



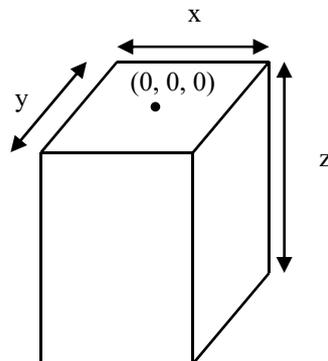
関節の角度



進行方向

部位の位置関係(正面から見た図)

関数myBoxの仕様



void myBox(double x, double y, double z)

プロトタイプ

```
#include <stdlib.h>
#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <GL/glut.h>

int winw, winh;
const int delay = 50;
const double WALKSPEED[2] = { 0.1, 0.05 };
const double WALKAREA[2] = { 3.0, 2.0 };
const double WALKTURN[2] = { 90.0, -90.0 };
double dir[2], posx[2], posy[2];
int timestep, cycle[2];

void myBox(double x, double y, double z)
{
    glBegin(GL_QUADS);

    glNormal3d(0.0, 0.0, 1.0);
    glVertex3d(-x / 2.0, -y / 2.0, 0.0);
    glVertex3d(x / 2.0, -y / 2.0, 0.0);
    glVertex3d(x / 2.0, y / 2.0, 0.0);
    glVertex3d(-x / 2.0, y / 2.0, 0.0);

    glNormal3d(0.0, 0.0, -1.0);
    glVertex3d(-x / 2.0, y / 2.0, -z);
    glVertex3d(x / 2.0, y / 2.0, -z);
    glVertex3d(x / 2.0, -y / 2.0, -z);
    glVertex3d(-x / 2.0, -y / 2.0, -z);

    glNormal3d(1.0, 0.0, 0.0);
    glVertex3d(x / 2.0, -y / 2.0, 0.0);
    glVertex3d(x / 2.0, -y / 2.0, -z);
    glVertex3d(x / 2.0, y / 2.0, -z);
    glVertex3d(x / 2.0, y / 2.0, 0.0);

    glNormal3d(0.0, 1.0, 0.0);
    glVertex3d(x / 2.0, y / 2.0, 0.0);
    glVertex3d(x / 2.0, y / 2.0, -z);
    glVertex3d(-x / 2.0, y / 2.0, -z);
    glVertex3d(-x / 2.0, y / 2.0, 0.0);

    glNormal3d(-1.0, 0.0, 0.0);
    glVertex3d(-x / 2.0, y / 2.0, 0.0);
    glVertex3d(-x / 2.0, y / 2.0, -z);
    glVertex3d(-x / 2.0, -y / 2.0, -z);
    glVertex3d(-x / 2.0, -y / 2.0, 0.0);

    glNormal3d(0.0, -1.0, 0.0);
    glVertex3d(-x / 2.0, -y / 2.0, 0.0);
    glVertex3d(-x / 2.0, -y / 2.0, -z);
    glVertex3d(x / 2.0, -y / 2.0, -z);
    glVertex3d(x / 2.0, -y / 2.0, 0.0);

    glEnd();
}
```

プロトタイプ

```
void myHuman(int t, int c) {
    double leglright, leglleft, leg2right, leg2left;
    double armlright, armlleft, arm2right, arm2left;
    double a = (double)(t % c) / (double)c;

    leglright = 0.0; // 【課題】 角度を適切に計算
    leglleft = 0.0; // 【課題】 角度を適切に計算
    leg2right = 0.0; // 【課題】 角度を適切に計算
    leg2left = 0.0; // 【課題】 角度を適切に計算
    armlleft = 0.0; // 【課題】 角度を適切に計算
    armlright = 0.0; // 【課題】 角度を適切に計算
    arm2left = 0.0; // 【課題】 角度を適切に計算
    arm2right = 0.0; // 【課題】 角度を適切に計算

    // 【課題】 ここから

    myBox(0.15, 0.15, 0.25); // head

    glTranslated(0.0, 0.0, -0.3);
    myBox(0.45, 0.25, 0.65); // body

    glTranslated(0.3, 0.0, 0.0);
    glRotated(armlright, 1.0, 0.0, 0.0);
    myBox(0.15, 0.15, 0.35); // arm right shoulder

    glTranslated(0.0, 0.0, -0.4);
    glRotated(arm2right, 1.0, 0.0, 0.0);
    myBox(0.15, 0.15, 0.35); // arm right elbow

    glTranslated(-0.3, 0.0, 0.0);
    glRotated(armlleft, 1.0, 0.0, 0.0);
    myBox(0.15, 0.15, 0.35); // arm left shoulder

    glTranslated(0.0, 0.0, -0.4);
    glRotated(arm2left, 1.0, 0.0, 0.0);
    myBox(0.15, 0.15, 0.35); // arm left elbow

    glTranslated(0.125, 0.0, -0.7);
    glRotated(leglright, 1.0, 0.0, 0.0);
    myBox(0.2, 0.2, 0.4); // leg right hip

    glTranslated(0.0, 0.0, -0.45);
    glRotated(leg2right, 1.0, 0.0, 0.0);
    myBox(0.2, 0.2, 0.4); // leg right knee

    glTranslated(-0.125, 0.0, -0.7);
    glRotated(leglleft, 1.0, 0.0, 0.0);
    myBox(0.2, 0.2, 0.4); // leg left hip

    glTranslated(0.0, 0.0, -0.45);
    glRotated(leg2left, 1.0, 0.0, 0.0);
    myBox(0.2, 0.2, 0.4); // leg left knee

    // 【課題】 ここまで
    // glPushMatrixやglPopMatrixを適切に追加
}
```

プロトタイプ

```
void myGround()
{
    float ground[2][4] = {
        {0.6f, 0.6f, 0.6f, 1.0f},
        {0.3f, 0.3f, 0.3f, 1.0f}
    };
    int i, j;

    glBegin(GL_QUADS);
    glNormal3d(0.0, 0.0, 1.0);
    for (j = -5; j < 5; j++) {
        for (i = -5; i < 5; i++) {
            glMaterialfv(GL_FRONT, GL_DIFFUSE, ground[abs(i + j) % 2]);
            glVertex3d((double)i, (double)j, 0.0);
            glVertex3d((double)(i + 1), (double)j, 0.0);
            glVertex3d((double)(i + 1), (double)(j + 1), 0.0);
            glVertex3d((double)i, (double)(j + 1), 0.0);
        }
    }
    glEnd();
}

void myDisplay()
{
    float red[4] = { 0.8f, 0.2f, 0.2f, 1.0f };
    float blue[4] = { 0.2f, 0.2f, 0.8f, 1.0f };

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (double)winw / (double)winh, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, -7.0, 7.0, 0.0, -1.0, 0.0, 0.0, 0.0, 1.0);

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glPushMatrix();

    myGround();

    glMaterialfv(GL_FRONT, GL_DIFFUSE, red);
    glPushMatrix();
    glTranslated(posx[0], posy[0], 1.9);
    glRotated(dir[0], 0.0, 0.0, 1.0);
    myHuman(timestep, cycle[0]);
    glPopMatrix();

    glMaterialfv(GL_FRONT, GL_DIFFUSE, blue);
    glPushMatrix();
    glTranslated(posx[1], posy[1], 1.9);
    glRotated(dir[1], 0.0, 0.0, 1.0);
    myHuman(timestep, cycle[1]);
    glPopMatrix();

    glPopMatrix();

    glutSwapBuffers();
}
```

プロトタイプ

```
void myTimer(int value)
{
    const double SMALLVAL = 0.01;
    double speedx, speedy;
    int i;

    if (value == 1) {
        timestep++;

        for (i = 0; i < 2; i++) {
            speedx = WALKSPEED[i] * cos((dir[i] + 90.0) / 180.0 * M_PI);
            speedy = WALKSPEED[i] * sin((dir[i] + 90.0) / 180.0 * M_PI);
            posx[i] += speedx;
            posy[i] += speedy;

            if (posx[i] > WALKAREA[i] + SMALLVAL) {
                posx[i] = WALKAREA[i];
                dir[i] += WALKTURN[i];
            }
            if (posy[i] > WALKAREA[i] + SMALLVAL) {
                posy[i] = WALKAREA[i];
                dir[i] += WALKTURN[i];
            }
            if (posx[i] < -WALKAREA[i] - SMALLVAL) {
                posx[i] = -WALKAREA[i];
                dir[i] += WALKTURN[i];
            }
            if (posy[i] < -WALKAREA[i] - SMALLVAL) {
                posy[i] = -WALKAREA[i];
                dir[i] += WALKTURN[i];
            }
        }

        glutTimerFunc(delay, myTimer, 1);
        glutPostRedisplay();
    }
}

void myKeyboard(unsigned char key, int x, int y)
{
    if (key == 0x1B) exit(0);
}

void myReshape(int width, int height)
{
    winw = width;
    winh = height;
    glViewport(0, 0, winw, winh);
}

void myInit(char* progname)
{
    winw = 500;
    winh = 500;
    timestep = 0;
    dir[0] = -90.0;
    posx[0] = -3.0;
    posy[0] = -3.0;
    cycle[0] = 20;
    dir[1] = 0.0;
    posx[1] = -2.0;
    posy[1] = -2.0;
    cycle[1] = 40;
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(winw, winh);
    glutInitWindowPosition(0, 0);
    glutCreateWindow(progname);
}
```

プロトタイプ

```
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    myInit(argv[0]);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(delay, myTimer, 1);
    glutReshapeFunc(myReshape);
    glutDisplayFunc(myDisplay);
    glutMainLoop();
    return 0;
}
```

